UNIVERSITAT POLITECNICA DE CATALUNYA

PARALLELISM

Roger Vilaseca Darne and Xavier Martin Ballesteros
PAR4110

Lab 2: Brief tutorial on OpenMP programming model

20th March 2019, Q1

Contents

1 Introduction

2 OpenMP questionnaire

2.1 Parallel regions

2.1.1 1.hello.c

1. How many times will you see the "Hello world!" message if the program is executed with "./1.hello"?

2. Without changing the program, how to make it to print 4 times the "Hello World!" message?

2.1.2 2.hello.c

1. Is the execution of the program correct? (i.e., prints a sequence of "(Thid) Hello (Thid) world!" being Thid the thread identifier). If not, add a data sharing clause to make it correct?

2. Are the lines always printed in the same order? Why the messages sometimes appear intermixed? (Execute several times in order to see this).

2.1.3 3.how many.c

Assuming the OMP NUM THREADS variable is set to 8 with "export OMP NUM THREADS=8"

1. How many "Hello world ..." lines are printed on the screen?

2. What does omp get num threads return when invoked outside and inside a parallel region?

2.1.4 4.data sharing.c

1. Which is the value of variable x after the execution of each parallel region with different datasharing attribute (shared, private, firstprivate and reduction)? Is that the value you would expect? (Execute several times if necessary)

2.2 Loop parallelism

2.2.1 1.schedule.c

1. Which iterations of the loops are executed by each thread for each schedule kind?

2.2.2 2.nowait.c

1. Which could be a possible sequence of printf when executing the program?

2. How does the sequence of printf change if the nowait clause is removed from the first for directive?

3. What would happen if dynamic is changed to static in the schedule in both loops? (keeping the nowait clause)

2.2.3 3.collapse.c

1. Which iterations of the loop are executed by each thread when the collapse clause is used?

2. Is the execution correct if the collapse clause is removed? Which clause (different than collapse) should be added to make it correct?

2.3 Synchronization

2.3.1 1.datarace.c

(execute several times before answering the questions)

1. Is the program always executing correctly?

2. Add two alternative directives to make it correct. Explain why they make the execution correct.

2.3.2 2.barrier.c

1. Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order?

2.3.3 3.ordered.c

1. Can you explain the order in which the "Outside" and "Inside" messages are printed?

2. How can you ensure that a thread always executes two consecutive iterations in order during the execution of the ordered part of the loop body?

2.4 Tasks

2.4.1 1.single.c

1. Can you explain why all threads contribute to the execution of instances of the single worksharing construct? Why are those instances appear to be executed in bursts?

2.4.2 2.fibtasks.c

1. Why all tasks are created and executed by the same thread? In other words, why the program is not executing in parallel?

2. Modify the code so that the program correctly executes in parallel, returning the same answer that the sequential execution would return.

2.4.3 3.synchtasks.c

1. Draw the task dependence graph that is specified in this program

The **depend** clause establish dependences, only between tasks, on the scheduling of tasks. There are different types of dependence:

- **in**: The task has to wait until all previously generated sibling tasks that reference at least one of the list items in an *out* or *inout* list end their execution.
- **out** and **inout**: The task has to wait untill all previously generated sibling tasks that reference at least one of the list items in an *in*, *out* or *inout* list end their execution.

In the given code, the first task has an **out** depende type, with only the variable a in its list. As there are no previously generated sibling tasks, this first task is not dependent of any task. Tasks 2 and 3 are not dependent of any previously generated sibling task. Although there exist some previously sibling tasks, none of them has variables b or c in its lists.

On the other hand, task 4 has an **in** dependence type with variables a and b and an **out** dependence type with variable d. Thus, this task will have to wait until tasks 1 and 2 end their execution.

Finally, task 5 has an **in** dependence type with variables c and d. This taks will depend from tasks 3 and 4. Hence, the task dependence graph is the following:

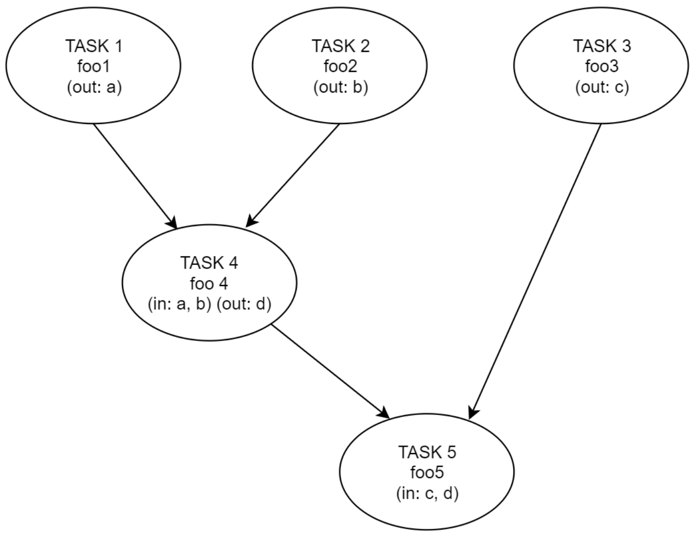

Figure 1: Task dependency graph of code 3.synchtasks.c.

2. Rewrite the program using only taskwait as task synchronisation mechanism (no depend clauses allowed)

The **taskwait** construct makes a task wait until the completion of its child tasks. The new version of the code is shown below. Task 4 will wait until tasks 1 and 2 end their execution and task 5 will wait until the completition of tasks 3 and 4. Doing this we recreated the task dependency graph of Figure 1 without using the **depend** clause.

The first problem we have seen is that task 3 cannot be executed at the same time than tasks 1 and 2. Otherwise, task 4 would have to wait until tasks 1, 2 and 3 end their execution altough it does not depend from task 3. The second problem is that with this code, we are not able to create the 5 tasks at the same time. Tasks 1, 2 and 4 will be created first. Then tasks 3 and 5.

```
    printf("Creating task foo1\n");
    #pragma omp task
    foo1();

    printf("Creating task foo2\n");
    #pragma omp task
    foo2();

    printf("Creating task foo4\n");
    #pragma omp taskwait
    foo4();

    printf("Creating task foo3\n");
    #pragma omp task
    foo3();

    printf("Creating task foo5\n");
    #pragma omp taskwait
    foo5();
```

Figure 2: Modified version of the given code.

Figures 3 and 4 show the output when executing the two versions of the code. We can see clearly the differences between one and the other explained before.

```
    Creating task foo1
    Creating task foo2
    Creating task foo3
    Creating task foo4
    Creating task foo5
    Starting function foo3
    Starting function foo1
    Starting function foo2
    Terminating function foo1
    Terminating function foo2
    Starting function foo4
    Terminating function foo4
    Terminating function foo3
    Starting function foo5
    Terminating function foo5
```

Figure 3: Output of the original version of the code.

```
    Creating task foo1
    Creating task foo2
    Creating task foo4
    Starting function foo2
    Starting function foo1
    Terminating function foo2
    Terminating function foo1
    Starting function foo4
    Terminating function foo4
    Creating task foo3
    Creating task foo5
    Starting function foo3
    Terminating function foo3
    Starting function foo5
    Terminating function foo5
```

Figure 4: Output of the modified version of the code.

2.4.4 4.taskloop.c

1. Find out how many tasks and how many iterations each task execute when using the grainsize and num tasks clause in a taskloop. You will probably have to execute the program several times in order to have a clear answer to this question.

The **grainsize(grain-size)** clause controls how many loop iterations are assigned to each created task. The number of loop iterations assigned to each created task is greater than or equal to the minimum of the value of grain-size and the total number of iterations.

We though that when using the **grainsize(5)** clause there will be created 3 tasks, the first two executing 5 iterations and the other one 2. In reality, there are only 2 tasks created, each one executing 6 iterations. This is because the number 5 does not mean that each task has 5 iterations but a minimum of 5, unless there remain less than 5 iterations.

The **num_tasks(num-tasks)** clause creates as many tasks as the minimum of num-tasks and the number of loop iterations.

There are 5 tasks created. The first two execute 3 iterations of the loop whereas the other 3 execute only 2.

```
Going to distribute 12 iterations with grainsize(5) ...
Loop 1: (1) gets iteration 0
Loop 1: (1) gets iteration 1
Loop 1: (1) gets iteration 2
Loop 1: (1) gets iteration 3
Loop 1: (2) gets iteration 6
Loop 1: (2) gets iteration 7
Loop 1: (2) gets iteration 8
Loop 1: (2) gets iteration 9
Loop 1: (2) gets iteration 10
Loop 1: (2) gets iteration 11
Loop 1: (1) gets iteration 4
Loop 1: (1) gets iteration 5
Going to distribute 12 iterations with num_tasks(5) ...
Loop 2: (2) gets iteration 0
Loop 2: (2) gets iteration 1
Loop 2: (1) gets iteration 6
Loop 2: (1) gets iteration 7
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
Loop 2: (1) gets iteration 8
Loop 2: (1) gets iteration 9
Loop 2: (2) gets iteration 2
Loop 2: (3) gets iteration 3
Loop 2: (3) gets iteration 4
Loop 2: (3) gets iteration 5
```

Figure 5: Ouput of the code without the nogroup clause.

2. What does occur if the nogroup clause in the first taskloop is uncommented?

The **taskloop** clause implicitly generates a **taskgroup** region that encloses it. In this region there is a barrier at the end, so the second loop will not be executed until the first one terminates. The **nogroup** clause removes the **taskgroup** region.

With the **nogroup** clause, the second loop can be executed at the same time than the first loop.

Now, there are created the same number of tasks, and each one does the same itreations than in the previous question. However, while two threads execute the first loop, the other two threads will execute the 5 tasks of the second loop (Figure 6), unless one of the two threads of the first loop end before the second loop terminates. In this situation, this thread will also execute tasks of the second loop (Figure 7).

```
    Going to distribute 12 iterations with grainsize(5) ...
    Going to distribute 12 iterations with num_tasks(5) ...
    Loop 1: (1) gets iteration 0
    Loop 1: (1) gets iteration 1
    Loop 1: (2) gets iteration 6
    Loop 1: (2) gets iteration 7
    Loop 1: (2) gets iteration 8
    Loop 1: (1) gets iteration 2
    Loop 2: (0) gets iteration 10
    Loop 2: (0) gets iteration 11
    Loop 2: (0) gets iteration 8
    Loop 2: (3) gets iteration 0
    Loop 2: (3) gets iteration 1
    Loop 2: (3) gets iteration 2
    Loop 2: (3) gets iteration 3
    Loop 2: (3) gets iteration 4
    Loop 2: (3) gets iteration 5
    Loop 2: (3) gets iteration 6
    Loop 2: (3) gets iteration 7
    Loop 1: (1) gets iteration 3
    Loop 1: (1) gets iteration 4
    Loop 1: (1) gets iteration 5
    Loop 2: (0) gets iteration 9
    Loop 1: (2) gets iteration 9
    Loop 1: (2) gets iteration 10
    Loop 1: (2) gets iteration 11
```

Figure 6: Ouput 1 of the code with the nogroup clause.

```
    Going to distribute 12 iterations with grainsize(5) ...
    Going to distribute 12 iterations with num_tasks(5) ...
    Loop 1: (2) gets iteration 0
    Loop 1: (2) gets iteration 1
    Loop 1: (2) gets iteration 2
    Loop 1: (2) gets iteration 3
    Loop 1: (2) gets iteration 4
    Loop 1: (2) gets iteration 5
    Loop 1: (1) gets iteration 6
    Loop 1: (1) gets iteration 7
    Loop 1: (1) gets iteration 8
    Loop 2: (0) gets iteration 10
    Loop 2: (3) gets iteration 3
    Loop 1: (1) gets iteration 9
    Loop 1: (1) gets iteration 10
    Loop 1: (1) gets iteration 11
    Loop 2: (0) gets iteration 11
    Loop 2: (0) gets iteration 8
    Loop 2: (0) gets iteration 9
    Loop 2: (3) gets iteration 4
    Loop 2: (3) gets iteration 5
    Loop 2: (1) gets iteration 6
    Loop 2: (1) gets iteration 7
    Loop 2: (2) gets iteration 0
    Loop 2: (2) gets iteration 1
    Loop 2: (2) gets iteration 2
```

Figure 7: Ouput 2 of the code with the nogroup clause.

3 Observing overheads